# SOFTVERSKO INŽINJERSTVO

## NAPREDNE TEHNIKE PROGRAMIRANJA

nevzudin.buzadjija@unze.ba

# Dobre i loše strane rekurzije

## Dobre strane rekurzije

Kod je (obično):

- kratak, čitljiv i jednostavan za razumijevanje,
- jednostavan za održavanje i otklanjanje grešaka,
- pogodan za dokazivanje korektnosti,
- ...

## Rekurziju treba koristiti:

- ako je rekurzivno rješenje "prirodno" i jednostavno za razumijevanje,
- ako rekurzivno rješenje ne zahtijeva suvišna izračunavanja koja je teško eliminisati,
- ako je ekvivalentno iterativno (nerekurzivno) rješenje previše kompleksno.

## Loše strane rekurzije

- Cijena poziva:
  - svaki rekurzivni korak znači novi stek okvir i kopiranje argumenata na stek, što dalje znači novo memorijsko zauzeće i usporavanje izvršavanja
  - u slučaju "dubokih" rekurzija, prostorna (memorijska) i vremenska složenost mogu biti kritične
- Suvišna izračunavanja:
  - svođenje složenog problema na jednostavnije može da rezultuje suvišnim ponavljanjima istih izračunavanja

# Dobre i loše strane rekurzije

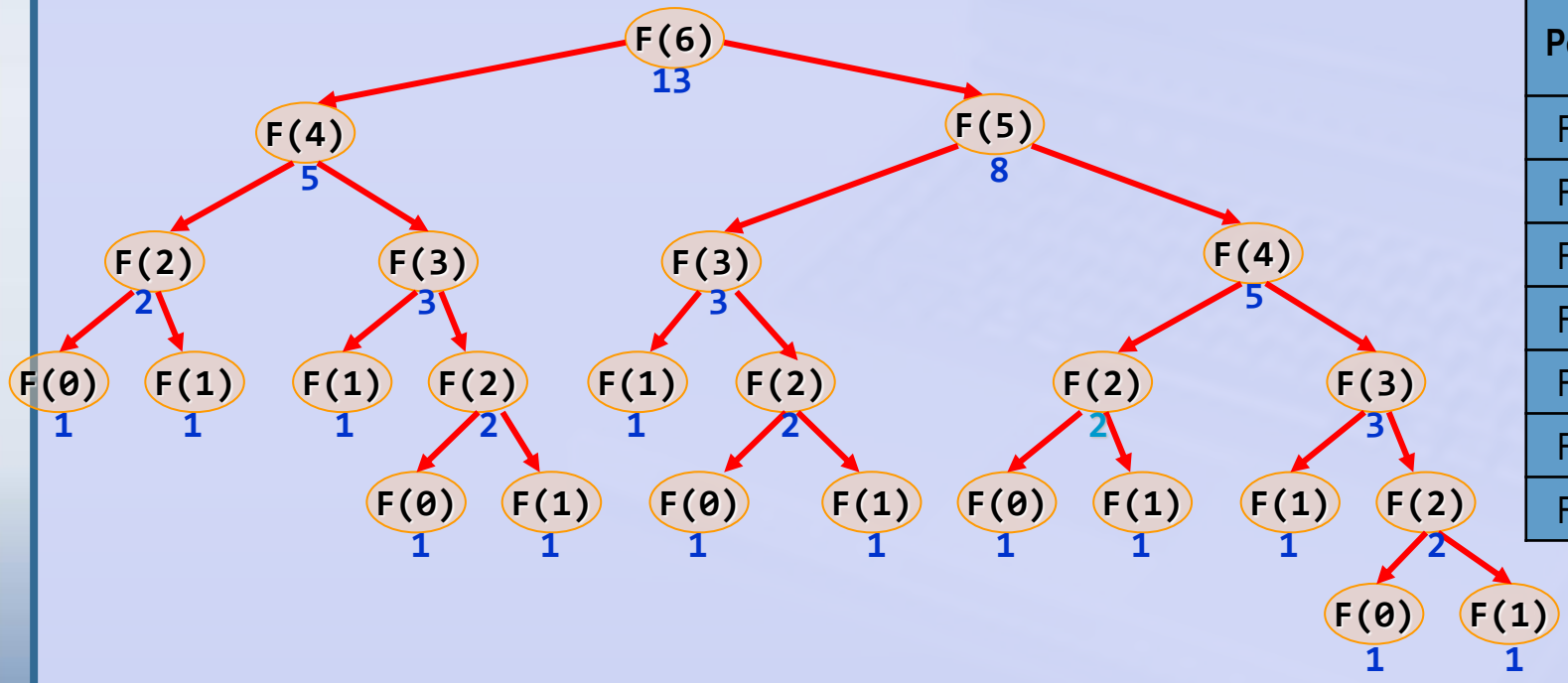
## Primjer suvišnih izračunavanja (Fibonačijev niz):

1, 1, 2, 3, 5, 8, 13, 21, 34,... (?)

$F_0=F_1=1$

$F_i=F_{i-1}+F_{i-2}$; $i>1$

```
int f(int i)
{
   if (i<=1) return 1;
   else return f(i-1)+f(i-2);
}
```




| Poziv | Broj izvršavanja |
|---|---|
| F(6) | 1 |
| F(5) | 1 |
| F(4) | 2 |
| F(3) | 3 |
| F(2) | 5 |
| F(1) | 8 |
| F(0) | 5 |

# Dobre i loše strane rekurzije

## Eliminacija suvišnih izračunavanja (memoizacija):

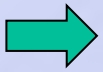
**Memoizacija je tehnika koja podrazumijeva pamćenje svih rezultata ranijih rekurzivnih poziva u odgovarajućoj strukturi podataka.**

**Prilikom ulaska u funkciju provjerava se da li je već izračunata tražena vrijednost.**

**Ako postoji izračunata vrijednost, vraća se rezultat.**

**Inače se izračunava nova vrijednost, dodaje u strukturu i vraća rezultat.**

```
int f(int i)
{
   if (i<=1) return 1;
   else return f(i-1)+f(i-2);
}
```



```
int f(int i)
{
   static int memo[MAX]={1,1};
   if (memo[i]) return memo[i];
   else return memo[i]=f(i-1)+f(i-2);
}
```

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| broj izvršavanja | 1 | 1 | 3 | 5 | 9 | 15 | 25 |

| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| min. broj izvršavanja | 1 | 1 | 3 | 3 | 3 | 3 | 3 |
| max. broj izvršavanja | 1 | 1 | 3 | 5 | 7 | 9 | 11 |

# Dobre i loše strane rekurzije

**Eliminacija suvišnih izračunavanja (redefinicija rekurzivnog koraka):**

$$x^n = \begin{cases} 1, & n = 0 \\ x \cdot x^{n-1}, & n > 0 \end{cases}$$

$$x^n = \begin{cases} 1, & n = 0 \\ (x \cdot x)^{n/2}, & n \text{ parno} \\ x \cdot x^{n-1}, & n \text{ neparno} \end{cases}$$

```
double stepenovanje(double x, int n)
{
   if (n==0)
      return 1;
   else
      return x*stepenovanje(x,n-1);
}
```

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| broj izvršavanja | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |

```
double stepenovanje(double x, int n)
{
   if (n==0)
      return 1;
   else
      if (n%2==0)
         return stepenovanje(x*x,n/2);
      else
         return x*stepenovanje(x,n-1);
}
```

| n | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|---|
| broj izvršavanja | 1 | 2 | 3 | 4 | 4 | 5 | 5 | 6 | 5 |

# Vrste rekurzivnih funkcija

**Postoje dvije vrste rekurzije**:

- tail rekurzija,
- non - tail rekurzija.

***Tail rekurzija*** jest situacija u kojoj se rekurzija događa na repu (*engl*. tail) funkcije, što znači da nakon rekurzivnog poziva nema drugih instrukcija.
Kao primjer *tail rekurzije* može se navesti brzo sortiranje (*engl*. Quick Sort).

***Non - tail recursion*** jest situacija u kojoj nakon poziva rekurzije postoje još druge instrukcije kao što je to slučaj kod sortiranja spajanjem (*engl*. Merge Sort).

# Eliminacija rekurzije

**Repni rekurzivni poziv** = rekurzivni poziv čiji je rezultat ujedno i rezultat funkcije, tj. nakon rekurzivnog poziva (i vraćanja rezultata) nema dodatnih naredbi/izračunavanja.

**Primjer:**

```
double stepenovanje(double x, int n)
{
   if (n==0)
      return 1;
   else
      if (n%2==0)
         return stepenovanje(x*x,n/2);
      else
         return x*stepenovanje(x,n-1);
}
```

repni rekurzivni poziv

nije repni rekurzivni poziv jer ima dodatno računanje nakon povratka iz pozvane funkcije

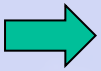
# Eliminacija repne rekurzije

**Repna rekurzija može da se eliminiše na sljedeći način:**

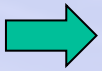
- **prije rekurzivnog poziva treba promijeniti argument tako da ima vrijednost koju bi imao kad se izvrši rekurzivni poziv,**
- nakon što se promijeni vrijednost argumenta, nema više potrebe da se vrši rekurzivni poziv nego je dovoljno **kontrolu prebaciti na početak funkcije** (npr. pomoću `goto`) ,
- refaktorisati kod tako da se **goto zamijeni odgovarajućom petljom**.

**Primjer:**

```
int f(int n)
{
   if (n==0)
      return 1;
   else
      f(n-1);
}
```



```
int f(int n)
{
   start:
   if (n==0)
      return 1;
   else
   {
      n=n-1;
      goto start;
   }
}
```



```
int f(int n)
{
   while (n>0)
      n=n-1;
   return 1;
}
```

# Eliminacija repne rekurzije

**Primjer (Euklidov algoritam za određivanje mjere dva broja):**

```
int mjera(int a, int b)
{
  if (b==0)
    return a;
  else
    return mjera(b,a%b);
}
```

```
int mjera(int a, int b)
{
  start:
  if (b==0)
    return a;
  else
  {
    int tmp=a%b;
    a=b;
    b=tmp;
    goto start;
  }
}
```

```
int mjera(int a, int b)
{
  while (b>0)
  {
    int tmp=a%b;
    a=b;
    b=tmp;
  }
  return a;
}
```

# Primjeri rekurzija

**Primjer (sekvencijalno pretraživanje niza):**

```
int search(tip niz[], tip x, int kapacitet, int i)
{
   if (i >= kapacitet) return -1;
   if (niz[i] == x) return i;
   return search(niz, x, kapacitet, i+1);
}
```

**Inicijalni poziv funkcije za pretraživanje**

```
search(niz,x,n,0)
```

**Primjer (poboljšano sekvencijalno pretraživanje niza sa stražom):**

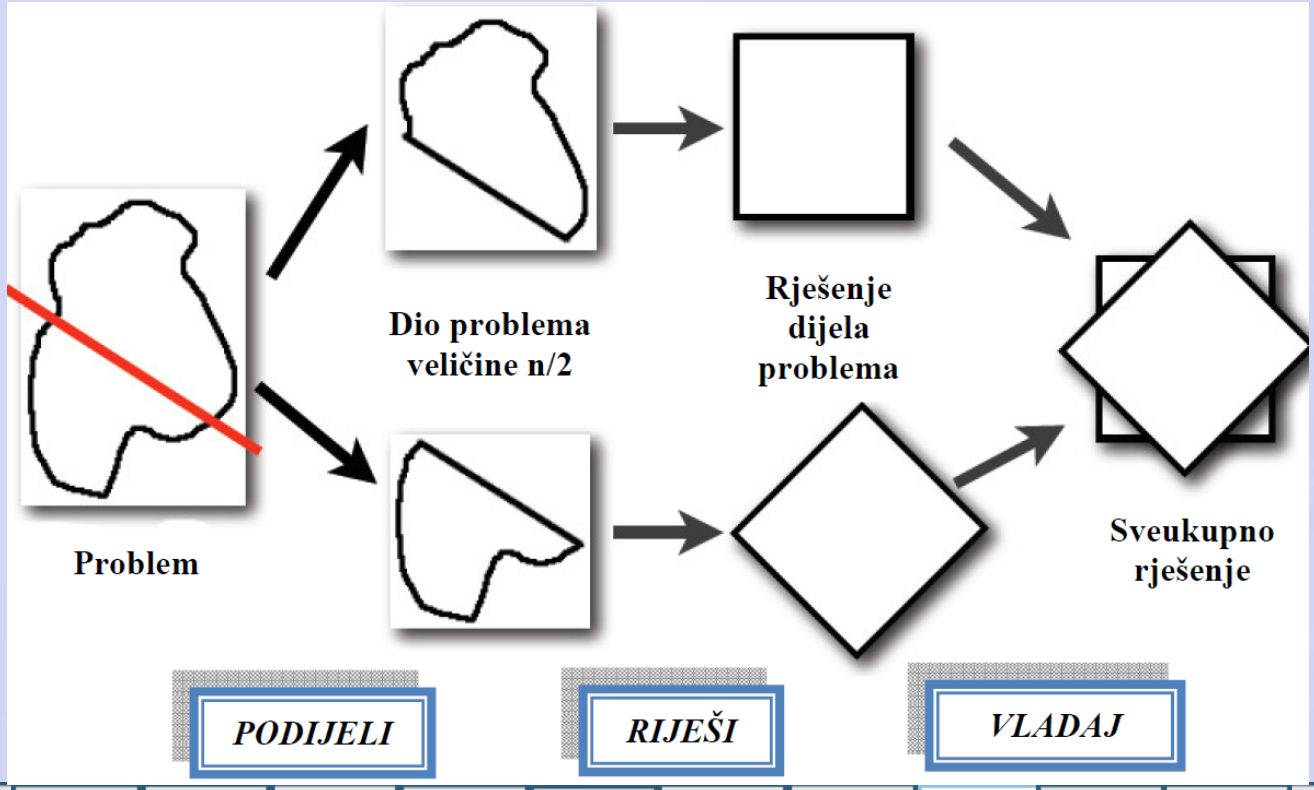
```
int search(tip niz[], tip x, int i)
{
   if (niz[i] == x) return i;
   return search(niz, x, i+1);
}
```

**Potreban kod u pozivaocu (na kraj niza dodaje se stražar – tražena vrijednost)**

```
niz[n]=x;
search(niz,x,0)
```

# Metoda *podijeli pa vladaj*

**Metoda *podijeli pa vladaj*** posebna je algoritamska tehnika u kojoj se ulaz razbija na nekoliko dijelova te se potom problemi u svakom dijelu rješavaju rekurzivno, a zatim se kombiniraju rješenja ovih manjih problema u konačno rješenje



U prvom koraku, ova metoda *teži* problem rastavlja na *lakše* probleme koji su slični polaznom problemu. Nakon toga rekurzija poziva samu sebe, ali ovaj put s tim *lakšim* problemom kao parametrom. Taj lakši problem dijeli se na još lakše i lakše, sve dok se ne dođe do osnovnog, trivijalnog problema koji znamo riješiti te tu dolazi do prekida rekurzije.

# Vrste metode *podijeli pa vladaj*

S obzirom na način kako se dijeli originalni problem, postoje dvije vrste metode *podijeli pa vladaj*. To su:

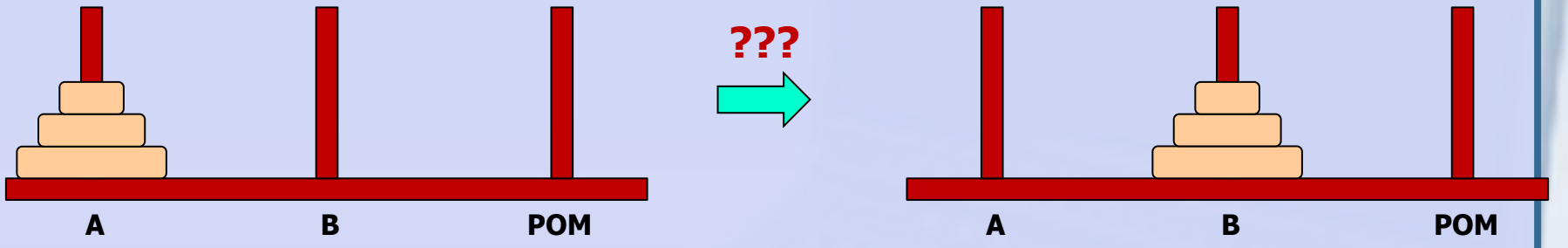
1. smanji pa vladaj
2. podijeli pa vladaj

Kod rješavanja problema podmetodom *smanji pa vladaj*, problem se rješava tako da se originalni problem smanji za neki faktor koji ovisno o problemu može biti konstantan ili varijabilan. Primjeri ove podmetode su *hanojski tornjevi*, *binarno pretraživanje*, itd.

Kod rješavanja problema podmetodom *podijeli pa vladaj*, problem se rješava tako da se originalni problem podijeli na *lakši* problem iste ili slične prirode te se tada pristupi rješavanju lakšeg problema. Primjeri ove podmetode su *Merge Sort*, *Quick Sort*, itd.

# Primjeri rekurzija

**Primjer (Hanojske kule – *Towers of Hanoi*):**




**Zadatak:**

**Prebaciti svih *n* (zlatnih) prstenova sa kule A na kulu B.**

**Pravila igre:**

1. **U jednom potezu može da se prebaci samo jedan prsten.**
2. **Manji prsten može da se stavi samo na veći prsten.**
3. **Za premještanje je dozvoljeno koristiti pomoćnu kulu POM.**

# Primjeri rekurzija

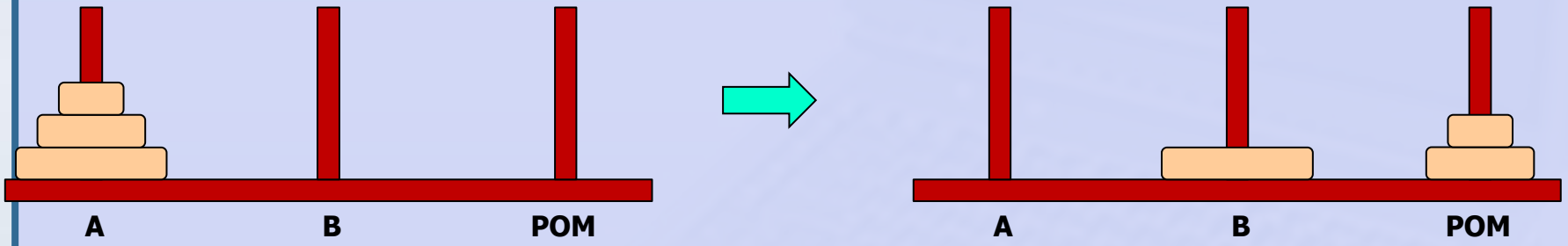
## Primjer (Hanojske kule – *Towers of Hanoi*):

**Ideja za rješavanje problema:**

- **REKURZIJA: problem prebacivanja *n* prstenova treba svesti na prebacivanje *n*-1 prstena.**

- **Ako prebacimo *n*-1 prstenova sa A na POM, tada ćemo moći preostali prsten prebaciti sa A na B.**

- **OSNOVNI SLUČAJ: za *n*=1, prsten se prebaci sa A na B**




- **Sada je najveći prsten na odgovarajućoj kuli (B) i problem je sveden sa *n* na *n*-1 prsten.**

- **Prebacivanje *n*-1 prstenova sa POM na B je isti problem kao i prebacivanje *n* prstenova sa A na B,**
  **samo jednostavniji (jer ima jedan prsten manje).**

- **Problem se svodi na PREBACIVANJE SA jedne kule NA drugu kulu PREKO trece kule**

**PREBACI(n, SA, NA, PREKO)**

# Primjeri rekurzija

**Primjer (Hanojske kule – *Towers of Hanoi*):**

**Algoritam: PREBACI(n,SA,NA,PREKO)**

```
1. Ako je n=1 ispisi SA->NA  (osnovni slučaj)
2. Inače
    2.1. PREBACI(n-1,SA,PREKO,NA)   (prebaci n-1, oslobodi najveći)
    2.2. ispisi SA->NA
    2.3. PREBACI(n-1,PREKO,NA,SA)   (prebaci preostalih n-1)
```

**Poziv algoritma:**

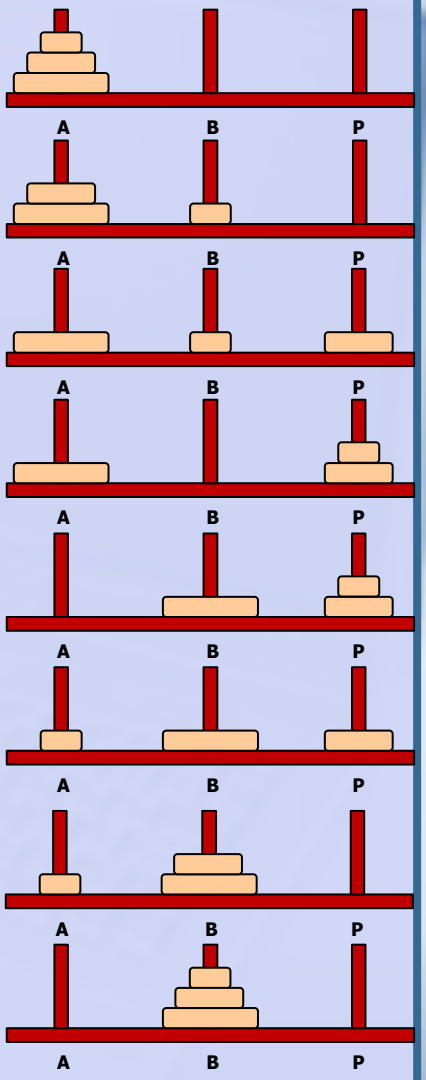
```
PREBACI(n, A, B, POM)
```

# Primjeri rekurzija

**Primjer (Hanojske kule – *Towers of Hanoi*):**

**Implementacija:**

```
void prebaci(int n, char sa, char na, char preko)
{
    if (n==1)
       cout<<sa<<na;
    else
    {
       prebaci(n-1,sa,preko,na);
       cout<<sa<<na;
       prebaci(n-1,preko,na,sa);
    }
}
int main()
{
   prebaci(3,'A','B','P');
   return 0;
}
```




**Rezultat izvršavanja:**

```
A->B A->P B->P A->B P->A P->B A->B
```

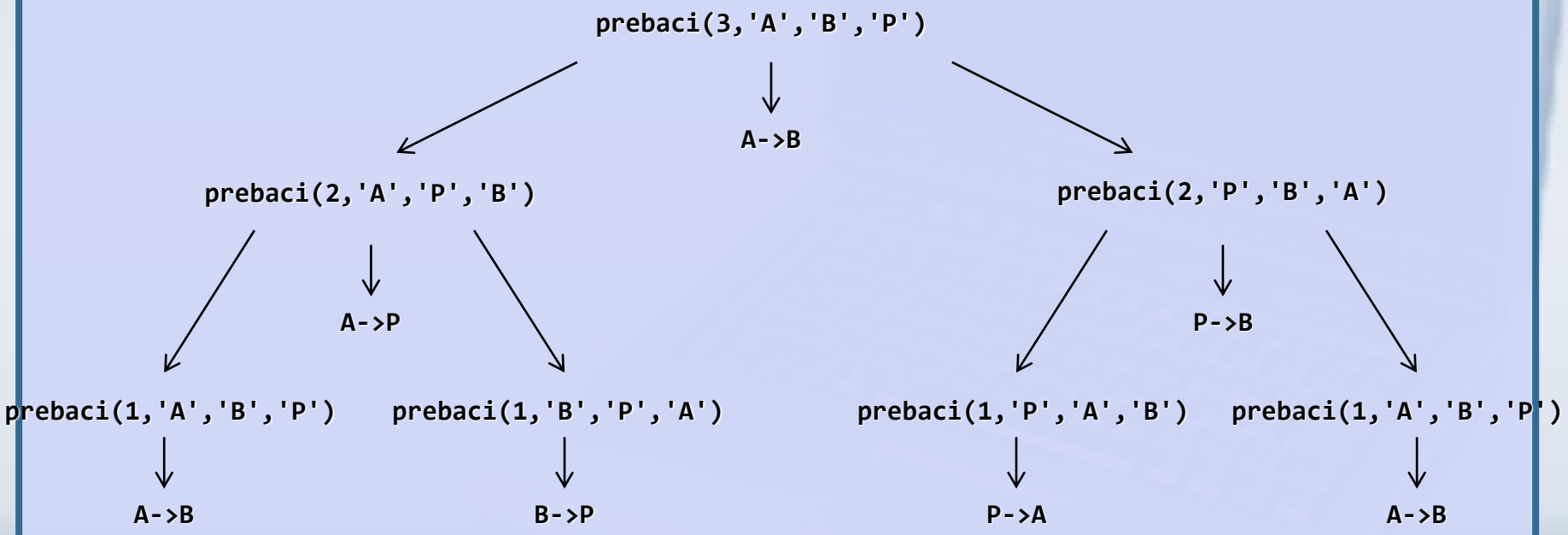
# Primjeri rekurzija

## Primjer (Hanojske kule – *Towers of Hanoi*):

**Rezultat izvršavanja:**

```
A->B A->P B->P A->B P->A P->B A->B
```

**Analiza izvršavanja (za *n*=3):**




| n | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|
| broj izvršavanja | 1 | 3 | 7 | 15 | 31 | 63 | 127 | 255 | 511 |

Funkcija će se zvati `pomakni_kulu`, a potrebni argumenti su: broj diskova koje treba pomaknuti, ime početnog tornja, ime ciljnog tornja te ime pomoćnog tornja.

```
void pomakni_kulu (int n, char A, char B,
char C){
if (n > 0){
pomakni_kulu (n-1, A, C, B); //1. pravilo
pomakni_disk (A, B); //2. pravilo
pomakni_kulu (n-1, C, B, A); //3. pravilo
}
}
```

Za slučaj kad postoji samo jedan disk, funkcija `pomakni_kulu` ne izvršava ništa pa se u tom slučaju izvršava funkcija `pomakni_disk(A, B)` koja je definirana na sljedeći način:

```
void pomakni_disk (char sa_kule, char na_kulu){
 cout<<"\t\t"<<sa_kule<<" -> "<<na_kulu<<endl;
                                }
```

```
#include <iostream>
using namespace std;
void pomakni_disk (char sa_kule, char na_kulu){
      cout<<"\t\t\t"<<sa_kule<<" -> "<<na_kulu<<endl;
        }

void pomakni_kulu (int n, char A, char B, char C){
if (n > 0){
pomakni_kulu (n-1, A, C, B);
pomakni_disk (A, B);
pomakni_kulu (n-1, C, B, A);
              }
}
```

```
int main (){
int n;
cout<<"\n\n\tBroj diskova: ";
cin>>n;
system("cls");
cout<<"\n\n\tSlijed poteza za "<<n<<" diskova: \n\n";
pomakni_kulu (n, 'A','B','C');
cout<<"\n\n";
system ("pause");
return 0;
}
```

1-org

2-nac

# Sortiranje spajanjem (Merge Sort)

Sortiranje spajanjem, poznatije još kao *Merge Sort*, jest algoritam sortiranja utemeljen na uspoređivanju.

Ovo sortiranje radi na principu da se dati niz koji treba sortirati podijeli u dva jednaka dijela te sortira svaku polovicu rekurzivno, a zatim kombinira rezultate te ih se spaja u jedan sortirani niz.

Koraci izvođenja ovog algoritma:
1. ako niz ima nulu ili jedan element, tad je on već sortiran. Inače,
2. nesortirani niz se dijeli u dva podniza približno jednake dužine,
3. rekurzivno se sortira svaki podniz ponovnom primjenom algoritma sortiranja,
4. spajaju se dva sortirana podniza u jedan sortirani niz.

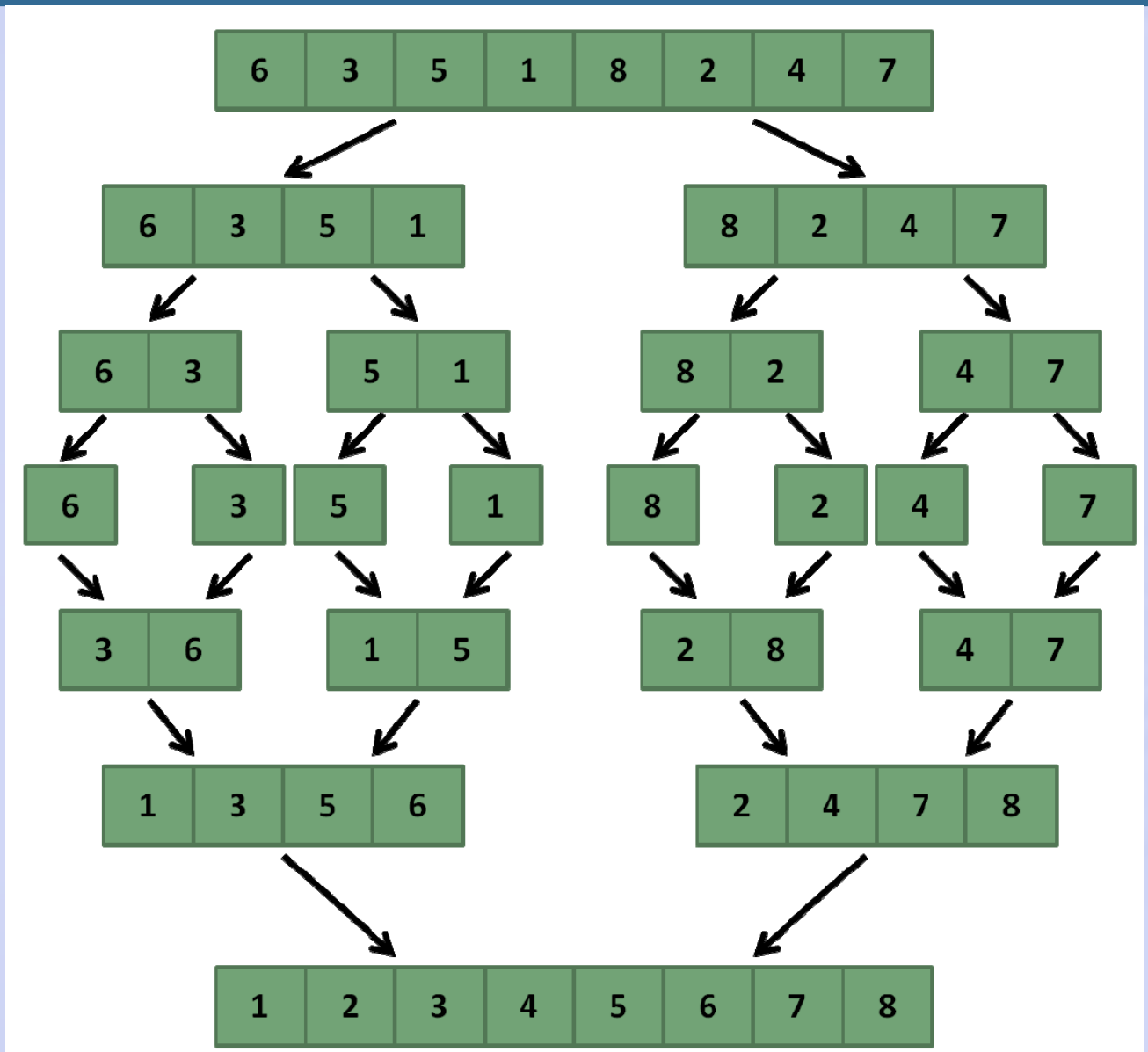
# Opći pseudokod sortiranja spajanjem glasi:

```
MergeSort(mali, veliki)
  ako je (mali<veliki)
  srednji=(mali+veliki)/2
  MergeSort(mali, srednji)
MergeSort(srednji+1, veliki)
Merge(mali, srednji veliki)
  kraj_MergeSort
```

No, osim same funkcije *MergeSort*, potrebna je i funkcija za spajanje čiji pesudokod glasi:

```
Merge(mali, srednji, veliki)
h=i=mali
j=srednji+1
sve dok je ((h<=srednji) i (j<=veliki)) ponavljaj
ako je (niz[h]<=niz[j])
b[i]=niz[h]
h++
inače
b[i]=niz[j]
j++
kraj ponavljanja
ako je (h>srednji)
```

```
za svaki k=j dok je k<=veliki ponavljaj
b[i]=niz[k]
i++
k++
kraj ponavljanja
inače
za svaki k=h dok je k<=srednji ponavljaj
b[i]=niz[k]
i++
k++
kraj ponavljanja
za svaki k=mali dok je k<=veliki ponavljaj
niz[k]=b[k]
kraj ponavljanja
kraj_Merge
```


6 3 5 1 8 2 4 7
6 3 5 1
8 2 4 7
6 3
5 1
8 2
4 7
6
3
5
1
8
2
4
7
3 6
1 5
2 8
4 7
1 3 5 6
2 4 7 8
1 2 3 4 5 6 7 8

```
#include <iostream>
#include <iomanip>
using namespace std;
int niz[1000];
void Merge(int mali, int srednji, int veliki){
    int h, i, j, k, b[1000];
    h=i=mali;
    j=srednji+1;
 while((h<=srednji) && (j<=veliki)){
        if(niz[h]<=niz[j]){
            b[i]=niz[h];
            h++;
            }
        else{
            b[i]=niz[j];
            j++;
             }
```

```
   i++;
}

if(h>srednji){
     for(k=j; k<=veliki; k++){
     b[i]=niz[k];
     i++;
     }
}
else{
     for(k=h; k<=srednji; k++){
     b[i]=niz[k];
     i++;
       }
}
```

```
for(k=mali; k<=veliki; k++)
    niz[k]=b[k];
}

void MergeSort(int mali, int veliki){
  int srednji;
  if (mali<veliki){
        srednji=(mali+veliki)/2;
        MergeSort(mali, srednji);
        MergeSort(srednji+1, veliki);
        Merge(mali, srednji, veliki);
   }
}
```

```
int main(){
int broj, i;
cout<<"\n\nUnesi broj elemenata u nizu: ";
cin>>broj;
cout<<"\n\nElementi niza su: \n\n";
//niz=new int[broj];
for(i=1; i<=broj; i++){
    niz[i]=rand()%1000+1;
    cout<<setw(5)<<niz[i];
}
cout<<"\n\n";
   MergeSort(1, broj);
   cout<<"\n\nSortirani niz: \n\n";
for(i=1; i<=broj; i++){
     cout<<setw(5)<<niz[i];
}
```

```
//delete [] niz;
cout<<"\n\n\n";
system("pause");
return 0;
}
```

**1- nac**

# Brzo sortiranje (Quick Sort)

Kao i prethodni algoritmi i ovaj je zasnovan na metodi *podijeli pa vladaj* na način da podijeli niz, odnosno listu, na dva podniza, odnosno dvije podliste. Quick Sort ima najbolji učinak ukoliko su podaci koji se nalaze u nizu, to jest koje je potrebno sortirati neureneni, a niz veliki.

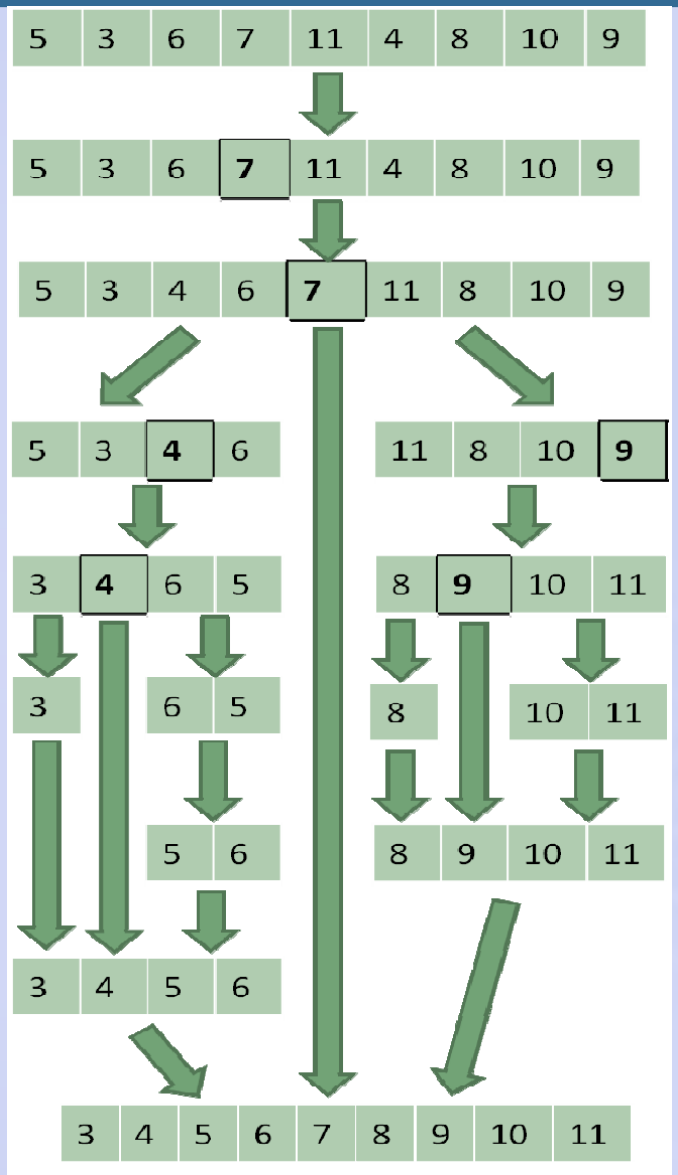
Koraci izvođenja ovog algoritma:

1. ako niz ima nulu ili jedan element, tada je on već sortiran. Inače,
2. niz se dijeli na dva dijela te se odabire stožerni element koji se još naziva *pivotom*,
3. niz se uredi na način da se svi elementi koji su manji od stožernog elementa stave ispred stožernog, a svi elementi koji su veći od stožernog stave iza stožernog,
4. rekurzivno se sortira podniz brojeva koji su manji od stožernog elementa, odnosno onih koji su veći.

Za bolje shvaćanje koraka slijedi opći pseudokod brzog sortiranja:

```
QuickSort(lijevi, desni, niz[])
i=lijevi, j=desni
pivot=niz[(lijevi+desni)/2]
ponavljanje
dok je (niz[i]<pivot)
i=i+1
dok je (pivot<niz[j])
j=j-1
ako je (i<=j)
temp=niz[i]
niz[i]=niz[j]
niz[j]=temp
i=i+1
```

```
j=j-1
sve dok je (i<=j)
ako je (lijevi<j)
QuickSort(lijevi, j, niz)
ako je (i<desni)
QuickSort(i, desni, niz)
kraj_QuickSort
```


5 3 6 7 11 4 8 10 9
5 3 6 7 11 4 8 10 9
5 3 4 6 7 11 8 10 9
5 3 4 6
11 8 10 9
3 4 6 5
8 9 10 11
3
6 5
8
10 11
5 6
8 9 10 11
3 4 5 6
3 4 5 6 7 8 9 10 11

```
#include <iostream>
#include <iomanip>
using namespace std;
int niz[1000];

void QuickSort(int lijevi, int desni, int niz[]){
int i=lijevi;
int j=desni;
int pivot=niz[(lijevi+desni)/2];
   while (i<=j){
          while(niz[i]<pivot)
              i=i+1;
          while(pivot<niz[j])
              j=j-1;
           if(i<=j){
```

```
            int temp=niz[i];
            niz[i]=niz[j];
            niz[j]=temp;
            i=i+1;
            j=j-1;
            }
    }
if (lijevi < j) QuickSort(lijevi, j, niz);
if (i < desni) QuickSort(i, desni, niz);
}
int main(){
int broj, i;
cout<<"\n\nUnesi broj elemenata u nizu: ";
cin>>broj;
cout<<"\n\nElementi niza su: \n\n";
```

```
            for(i=1; i<=broj; i++){
            niz[i]=rand()%1000+1;
            cout<<setw(5)<<niz[i];
            }
cout<<"\n\n";
            QuickSort(1, broj, niz);
cout<<"\n\nSortirani niz: \n\n";
            for(i=1; i<=broj; i++){
            cout<<setw(5)<<niz[i];
            }
cout<<"\n\n\n";
system("pause");
return 0;
}
```

**1-nac**

# Binarno pretraživanje (Binary Search)

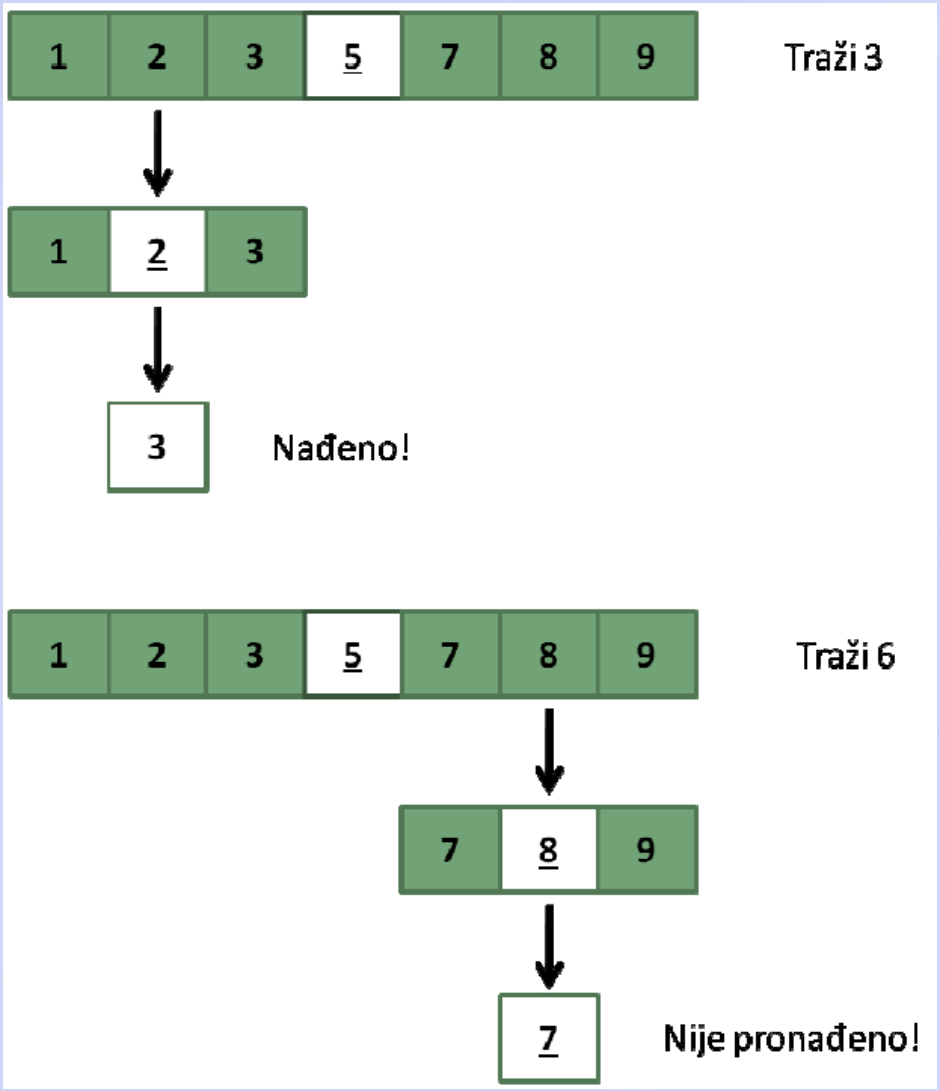
Binarno je pretraživanje algoritam za pretragu nekog niza ukoliko je taj niz već uzlazno sortiran. Ovo je jedan od najbržih načina pretraživanja niza koji je najbolje koristiti kod velikih nizova. On uzima srednji element sortiranog niza, dijeli niz na pola te, ukoliko je tražena vrijednost manja ili jednaka od tog srednjeg elementa, odbacuje desni podniz gdje se nalaze vrijednosti veće od srednjeg elementa te rekurzivno ponavlja postupak na podnizu gdje se nalaze vrijednosti manje od srednjeg elementa.

Koraci kod izvršavanja binarnog pretraživanja:

- pronalazi se srednji element,
- prema uslovu se odbacuje jedna polovica niza,
- pretražuje se druga polovica rekurzivno sve dok se traženi element ne pronađe ili dok ne ostane niti jedan element za pretraživanje.

Opći pseudokod binarnog pretraživanja glasi:

```
BinarySearch(niz[], pocetak, kraj, broj)
ako je (pocetak<=kraj)
srednji=(pocetak+kraj)/2
ako je (niz[srednji]==broj)
ispis: "Broj je pronanen!"
ako je (broj<niz[srednji])
kraj=srednji-1
BinarySearch(niz, pocetak, kraj, broj);
inače
pocetak=srednji+1
BinarySearch(niz, pocetak, kraj, broj);
kraj_BinarySearch
```


1 2 3 5 7 8 9
Traži 3
1 2 3
3
Nađeno!
1 2 3 5 7 8 9
Traži 6
7 8 9
7
Nije pronađeno!

```
#include <iostream>
#include <iomanip>
using namespace std;
void BinarySearch(int niz[15], int pocetak, int kraj, int broj){
int srednji, nasao;
        if(pocetak<=kraj){
            srednji=(pocetak+kraj)/2;
        if(niz[srednji]==broj){
            cout<<"\n\t\tBroj pronaden!\n";
nasao=1;
}
        if(broj<niz[srednji]){
            kraj=srednji-1;
            BinarySearch(niz, pocetak, kraj, broj);
        }
else{
```

```
            pocetak=srednji+1;
            BinarySearch(niz, pocetak, kraj, broj);
            }
    }
  else if (pocetak>kraj && nasao==0){
            cout<<"\n\t\tBroj nije pronaden!\n";
            }
}
int main(){
int niz[]={1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15};
int i, broj;
cout<<"\n\nElementi niza su: \n\n";
            for(i=0;i<15;i++){
            cout<<setw(5)<<niz[i];
            }
```

```
cout<<"\n\n\tTrazi: ";
cin>>broj;
        BinarySearch(niz, 0, 15, broj);
cout<<"\n\n";
system("pause");
return 0;
}
```

**1-nac**

# Primjer zadataka:

U matematici, **binomni koeficijent** je pozitivni cijeli broj, koji se pojavljuje kao koeficijent binomnog poučka. Indeksira se dvama ne-negativnim cijelim brojevima; binomni koeficijent s indeksima *n* i *k* obično se zapisuje kao:

$$\binom{n}{k}$$

i čita se *n* iznad ili povrh *k*. To je koeficijent člana $x^k$ polinomne ekspanzije binomne potencije oblika $(1+x)^n$. Pod odgovarajućim okolnostima vrijednost koeficijenta definirana je izrazom:

$$\frac{n!}{k!(n-k)!}$$

**Napisati progam koji ispisuje 25 clanova fibonacijevog niza.**

**Napisati rekurzivnu funkciju koja računa najveću cifru datog cijelog broja i program koji testira rad funkcije**

## Primjer zadataka:

**Napisati rekurzivnu funkciju koja prikazuje dekadne cifre datog cjelog broja. Napisati rekurzivnu funkciju koja prikazuje dekadne cifre datog cjelog broja u obrnutom poretku.**